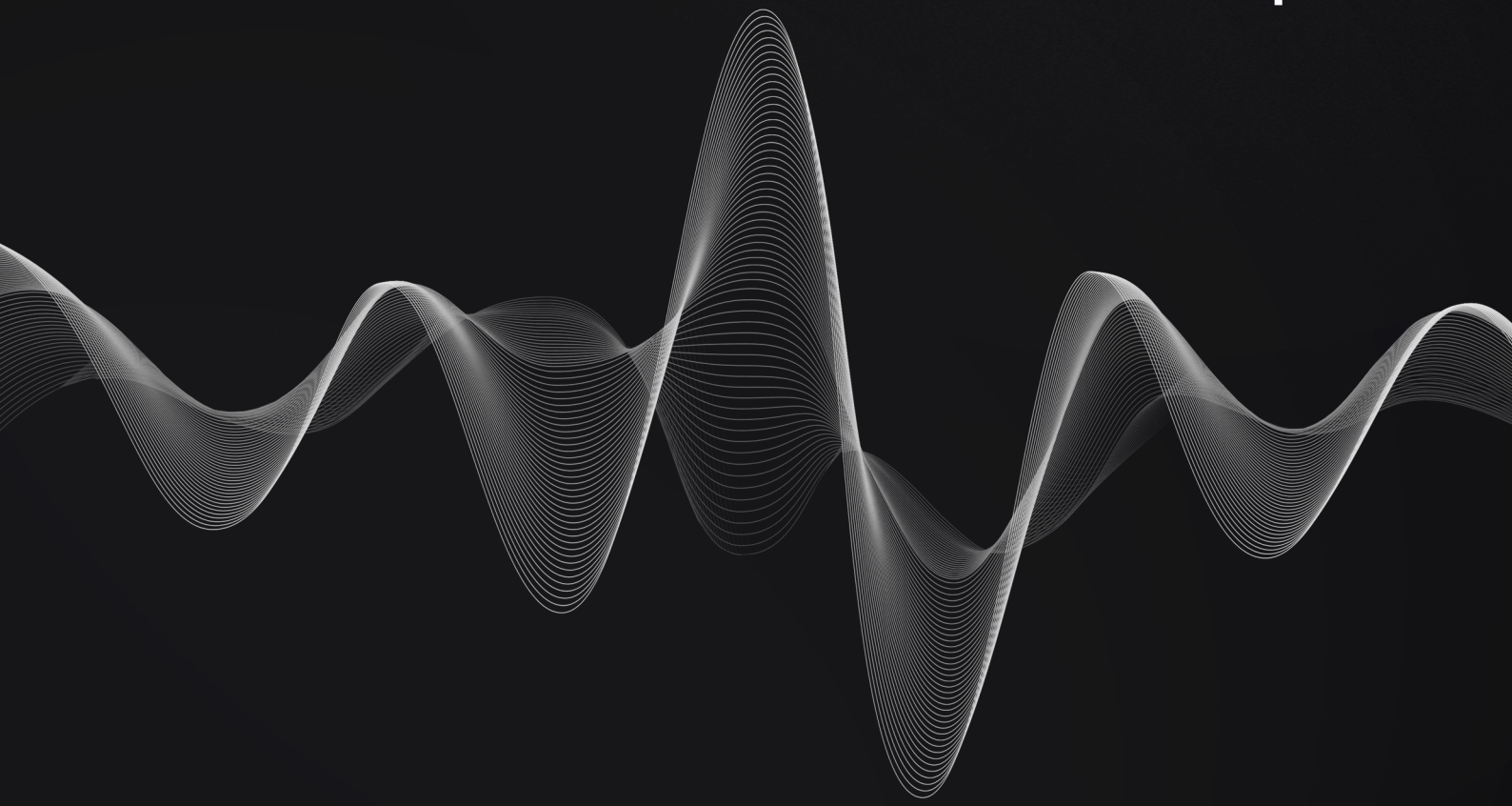
RESONANCE

primex

Primex Finance

Spot Margin Trading Protocol Audit Report



Blockchain, Emerging Technology, and Web2
CYBERSECURITY PRODUCT & SERVICE ADVISORY

Document Control

PUBLIC **FINAL**(v2.0)

Audit_Report_PRMX-TRP_FINAL_20

Jun 19, 2023	v0.1	Ilan Abitbol: Initial draft
Jun 21, 2023	v0.2	Michał Bazyli: Added findings
Jul 7, 2023	v0.3	João Simões: Final draft
Jul 10, 2023	v1.0	Charles Dray: Approved
Sep 27, 2023	v1.1	João Simões: Reviewed findings
Oct 18, 2023	v1.2	João Simões: Reviewed contract updates
Oct 19, 2023	v2.0	Charles Dray: Published

Points of Contact	Oleksandr Marukhnenko	Primex Finance	alex@primex.finance
	Charles Dray	Resonance	charles@resonance.security
	Luis Lubeck	Resonance	luis@resonance.security
Testing Team	Ilan Abitbol	Resonance	ilan.abitbol@resonance.security
	João Simões	Resonance	joao.simoes@resonance.security
	Michał Bazyli	Resonance	michal.bazyli@resonance.security

Copyright and Disclaimer

Contents

Executive Summary

Primex Finance contracted the services of Resonance to conduct a comprehensive security audit of their smart contracts **between June 19, 2023** and **July 10, 2023**. The primary objective of the assessment was to identify any potential security vulnerabilities and ensure the correct functioning of smart contract operations.

During the engagement, Resonance allocated **3** engineers to perform the security review. The engineers, including an accomplished professional with extensive proficiency in blockchain and smart-contract security, encompassing specialized skills in advanced penetration testing, and in-depth knowledge of multiple blockchain protocols, devoted **15 days** to the project. The project's test targets, overview, and coverage details are available throughout the next sections of the report.

The ultimate goal of the audit was to provide **Primex Finance** with a detailed summary of the findings, including any identified vulnerabilities, and recommendations to mitigate any discovered risks. The results of the audit are presented in detail further below.

System Overview

Primex Finance is a decentralized exchange platform powered by smart contract where users can lend and swap tokens, perform trading operations, and earn rewards by utilizing the protocol.

The system is composed of several components: the bucket infrastructure where all the lending and borrowing happens; the trading managers that implement logic to swap, open and close positions, with or without conditions; and the protocol management where Primex admin control emergency pauses, propositions and executions.

As a general workflow, a lender is able to stake collateral on a bucket and receive yield-bearing tokens. The collateral can be used by traders to borrow and open margin positions. The keepers handle the conditions that occur over time, such as, opening limit orders when price as reached the limit price, liquidating users with underwater positions, closing trades with set stop-losses, etc.

By using the different components of the protocol, all actors receive rewards that incentivise further usage. One of such rewards is the PMX token that offers utility and governance capabilities on the protocol.

Repository Coverage and Quality

Code	Tests	Documentation
7 / 10	8 / 10	7 / 10

Resonance's testing team has assessed the Code, Tests, and Documentation coverage and quality of the system and achieved the following results:

- The code follows development best practices and makes use of known patterns, standard libraries, and language guides. It is easily readable but does not use the latest stable version of relevant components. Overall, **code quality is good**.
- Unit and integration tests are included. The tests cover both technical and functional requirements. Code coverage is undetermined. Overall, **tests coverage and quality is good**.
- The documentation includes the specification of the system, technical details for the code, relevant explanations of workflows and interactions. Overall, **documentation coverage and quality is good**.

Target

The objective of this project is to conduct a comprehensive review and security analysis of the smart contracts that are contained within the specified repository.

The following items are included as targets of the security assessment:

- Repository (main): primex-finance/primex-protocol/src/contracts
- Hash: a8a22bcd2a84ad84b6b4644547183ce2d2412d15
- Repository: primex-finance/primex_contracts/src/contracts
- Hash: f809cc047193501369940 7dcd9eab63b60cd2e22

The following items are excluded:

- External and standard libraries;
- Files pertaining to the deployment process.

Methodology

In the context of security audits, Resonance's primary objective is to portray the workflow of a real-world cyber attack against an entity or organization, and document in a report the findings, vulnerabilities, and techniques used by malicious actors. While several approaches can be taken into consideration during the assessment, Resonance's core value comes from the ability to correlate automated and manual analysis of system components and reach a comprehensive understanding and awareness with the customer on security-related issues.

Resonance implements several and extensive verifications based off industry's standards, such as, identification and exploitation of security vulnerabilities both public and proprietary, static and dynamic testing of relevant workflows, adherence and knowledge of security best practices, assurance of system specifications and requirements, and more. Resonance's approach is therefore consistent, credible and essential, for customers to maintain a low degree of risk exposure.

Ultimately, product owners are able to analyze the audit from the perspective of a malicious actor and distinguish where, how, and why security gaps exist in their assets, and mitigate them in a timely fashion.

Source Code Review - Solidity EVM

During source code reviews for Web3 assets, Resonance includes a specific methodology that better attempts to effectively test the system in check:

1. Review specifications, documentation, and functionalities
2. Assert functionalities work as intended and specified
3. Deploy system in test environment and execute deployment processes and tests
4. Perform automated code review with public and proprietary tools
5. Perform manual code review with several experienced engineers
6. Attempt to discover and exploit security-related findings
7. Examine code quality and adherence to development and security best practices
8. Specify concise recommendations and action items
9. Revise mitigating efforts and validate the security of the system

Additionally and specifically for Solidity EVM audits, the following attack scenarios and tests are recreated by Resonance to guarantee the most thorough coverage of the codebase:

- Reentrancy attacks
- Frontrunning attacks
- Unsafe external calls
- Unsafe third party integrations
- Denial of service
- Access control issues

- Inaccurate business logic implementations
- Incorrect gas usage
- Arithmetic issues
- Unsafe callbacks
- Timestamp dependence
- Mishandled panics, errors and exceptions

Severity Rating

Security findings identified by Resonance are rated based on a Severity Rating which is, in turn, calculated off the **impact** and **likelihood** of a related security incident taking place. This rating provides a way to capture the principal characteristics of a finding in these two categories and produce a score reflecting its severity. The score can then be translated into a qualitative representation to help customers properly assess and prioritize their vulnerability management processes.

The **impact** of a finding can be categorized in the following levels:

1. Weak - Inconsequential or minimal damage or loss
2. Medium - Temporary or partial damage or loss
3. Strong - Significant or unrecoverable damage or loss

The **likelihood** of a finding can be categorized in the following levels:

1. Unlikely - Requires substantial knowledge or effort or uncontrollable conditions
2. Likely - Requires technical knowledge or no special conditions
3. Very Likely - Requires trivial knowledge or effort or no conditions

Impact \ Likelihood	Very Likely	Likely	Unlikely
Strong	Critical	High	Medium
Medium	High	Medium	Low
Weak	Medium	Low	Info

Repository Coverage and Quality Rating

The assessment of Code, Tests, and Documentation coverage and quality is one of many goals of Resonance to maintain a high-level of accountability and excellence in building the Web3 industry. In Resonance it is believed to be paramount that builders start off with a good supporting base, not only development-wise, but also with the different security aspects in mind. A product, well thought out and built right from the start, is inherently a more secure product, and has the potential to be a game-changer for Web3's new generation of blockchains, smart contracts, and dApps.

Accordingly, Resonance implements the evaluation of the code, the tests, and the documentation on a score **from 1 to 10** (1 being the lowest and 10 being the highest) to assess their quality and coverage. In more detail:

- Code should follow development best practices, including usage of known patterns, standard libraries, and language guides. It should be easily readable throughout its structure, completed with relevant comments, and make use of the latest stable version components, which most of the times are naturally more secure.
- Tests should always be included to assess both technical and functional requirements of the system. Unit testing alone does not provide sufficient knowledge about the correct functioning of the code. Integration tests are often where most security issues are found, and should always be included. Furthermore, the tests should cover the entirety of the codebase, making sure no line of code is left unchecked.
- Documentation should provide sufficient knowledge for the users of the system. It is useful for developers and power-users to understand the technical and specification details behind each section of the code, as well as, regular users who need to discern the different functional workflows to interact with the system.

Findings

During the security audit, several findings were identified to possess a certain degree of security-related weaknesses. These findings, represented by unique IDs, are detailed in this section with relevant information including Severity, Category, Status, Code Section, Description, and Recommendation. Further extensive information may be included in corresponding appendices should it be required.

An overview of all the identified findings is outlined in the table below, where they are sorted by Severity and include a **Remediation Priority** metric asserted by Resonance's Testing Team. This metric characterizes findings as follows:

- **"Quick Win"** Requires little work for a high impact on risk reduction.
- **"Standard Fix"** Requires an average amount of work to fully reduce the risk.
- **"Heavy Project"** Requires extensive work for a low impact on risk reduction.

RES-01	Possible To Drain Treasury On closePositionByCondition()		Resolved
RES-02	Lack of Penalty System For Liquidatable Traders		Acknowledged
RES-03	Oracle Data Feed Can Be Outdated Yet Used Anyways		Acknowledged
RES-04	Possible To Frontrun setBucket()		Resolved
RES-05	Missing Check For Initial Parameters In Bucket.sol		Resolved
RES-06	Bucket's Aave Liquidity May Be Temporarily Blocked		Acknowledged
RES-07	Possible To Activate Bucket Without Adding It		Resolved
RES-08	Rewards Can Be Claimed When LiquidityMiningRewardDistributor Is Paused		Resolved
RES-09	Possible To Use Deposit Owned By Other Users On depositProlongation()		Resolved
RES-10	Missing supportsInterface() Validation Of liquidityMiningRewardDistributor		Resolved
RES-11	Missing supportsInterface() Validation Of _bucketImplementation		Resolved
RES-12	Missing Validation Of Zero Amount On batchBurn()		Acknowledged
RES-13	Usage Of Unlimited Approve		Resolved

RES-14	Different Versions Of canBeFilled() Lead To Undefined Behaviour		Resolved
RES-15	Unnecessary Validation Of _params On createBucket()		Resolved
RES-16	Conditions On supportsInterface() Can Be Switched To Save Gas		Resolved

Possible To Drain Treasury On closePositionBy-Condition()

Critical **RES-PRMX-TRP01** Data Validation **Resolved**

Code Section

- contracts/PositionManager/PositionManager.sol#L233-255

Description

The function `closePositionByCondition()` is typically used by keepers to perform the upkeep of open positions that should be closed for a variety of reasons. This function was not designed to be used by traders that simply want to close a position without any condition. For that, the function `closePosition()` should be used.

However, the function `closePositionByCondition()` does not validate who is the caller, or the provided close reason. Therefore, it is possible for a trader to also use this function to close an active position without any condition. Despite the logic of the code being very similar between the two functions, since the function `closePositionByCondition()` was meant to be used by keepers, the keepers obtain rewards for performing the necessary upkeep.

Since a trader can also use the function `closePositionByCondition()` at will, there is a greater incentive for them to use it over `closePosition()`, since they will earn rewards for simply closing their active positions.

The rewards are claimable through the treasury, and with the possibility of this fact occurring over and over again by every trader, it is possible to drain the treasury of all its `PMX` and native tokens, therefore gaining an unfair advantage over other users not aware of this issue.

Recommendation

It is recommended to validate the caller and reasons provided to the `closePositionByCondition()` function to not allow traders to close positions with the reason `CLOSE_BY_TRADER`.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Lack of Penalty System For Liquidatable Traders

High | RES-PRMX-TRP02 | Business Logic | Acknowledged

Code Section

- contracts/libraries/PositionLibrary.sol#L388-402

Description

When a trader attempts to close a position, no validation is made on its health. This means that traders can close active positions at will even if the position is ready to be liquidated. While the Primex Protocol provides incentives for keepers to perform the necessary upkeep and liquidate positions that require so, traders can frontrun the keepers and deny them of their rewards.

Additionally, since there is no penalty system implemented for traders that close unhealthy positions with the reason `CLOSE_BY_TRADER`, this yields an inherent incentive to traders to take on the maximum amount of leverage on every trade. Adding to the fact that traders receive rewards through activity on the DEX, this problem could exacerbate, and impact the supply of `PMX` tokens.

Recommendation

It is recommended to implement a penalty system for traders that possess unhealthy positions and deny them the possibility of closing those positions with the reason `CLOSE_BY_TRADER`.

Status

The issue was acknowledged by Primex's team. The development team stated "The economic model of the protocol does not consider liquidation as a significant source of protocol revenue. When a position reaches the liquidation level, the trader starts competing with keepers to close that specific position. If the trader is faster than others, the protocol can enable them to retain the remaining deposit without exposing the protocol to any losses. We do not wish to prevent traders from closing their positions, as the main goal is to prevent lenders from incurring losses due to delayed liquidations.".

Oracle Data Feed Can Be Outdated Yet Used Anyways

Medium | RES-PRMX-TRP03 | Data Validation | Acknowledged

Code Section

- Not defined

Description

The Chainlink priceFeed is used in the PriceOracle.sol as a source of price for assets. However, `latestRoundData()` function of *priceFeed* returns `roundId`, `answer`, `startedAt`, `updatedAt`, `asnwerInRound` values but only the `answer` value is used in the functions. The retrieved price of the *priceFeed* can be outdated and used anyways as a valid data because no timestamp tolerance of the update source time is checked while storing the return parameters of `latestRoundData()`.

Recommendation

Adding a tolerance to compare the return timestamp of `updatedAt` from the `latestRoundData()` function with the current block timestamp is advisable. This helps to ensure that the `priceFeed` is updated according to the required frequency.

As Chainlink recommends :

Your application should track the `latestTimestamp` *variable or use the* `updatedAt` *value from the* `latestRoundData()` *function to make sure that the latest answer is recent enough for your application to use it. If your application detects that the reported answer is not updated within the heartbeat or within time limits that you determine are acceptable for your application, pause operation or switch to an alternate operation mode while identifying the cause of the delay.*

Status

The issue was acknowledged by Primex's team. The development team stated "Primex has an emergency pause system that allows EMERGENCY_ADMIN to pause borrowing funds from credit buckets and forbids opening new positions, etc. Each CL feed has a different heartbeat, and the normal heartbeat value is not stored on-chain, so the behavior of the oracle is monitored off-chain. If an inadequate price update frequency is detected, the corresponding components of the protocol will be paused until the oracle is stabilized. Furthermore, the team will provide users in the Primex app with information about outdated oracle prices. In future versions, the team is considering using reserve oracles for critical situations like these.".

Possible To Frontrun setBucket()

Medium | RES-PRMX-TRP04 | Data Validation | Resolved

Code Section

- contracts/DebtToken/DebtToken.sol#25-32
- contracts/PToken/PToken.sol#L26-33

Description

The function `setBucket()` is meant to be called right after the creation of a bucket. While the creation of buckets is handled by the contract `BucketsFactory.sol`, it is possible to create a bucket without using the factory. In these cases, the call to the `setBucket()` function is not guaranteed to happen on the same transaction. As such, it is possible for malicious actors to call such function right after the creation of a bucket and render the respective `PToken` and `DebtToken` useless by assigning them a malicious bucket.

Recommendation

It is recommended to implement proper validations to ensure that either buckets can only be created through the factory, or access control is implemented on `setBucket()` to only allow it to be called by the factory.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Missing Check For Initial Parameters In Bucket.sol

Medium | RES-PRMX-TRP05 | Data Validation | Resolved

Code Section

- contracts/Bucket/Bucket.sol#L72
- contracts/Bucket/Bucket.sol#L73
- contracts/Bucket/Bucket.sol#L74
- contracts/Bucket/Bucket.sol#L150
- contracts/Bucket/Bucket.sol#L156

Description

The Buckets contract is responsible for storing the liquidity provided by Lenders. Traders have the option to borrow this liquidity when they open margin positions. The deployment of the Bucket contract can only be done through the BucketFactory by `MEDIUM_TIMELOCK_ADMIN`.

During the initialization of the contract, most parameters are properly validated. However, there is a missing validation for the `withdrawalFeeRate` , `feeBuffer` and `reserveRate` As a result, there is a possibility that the bucket could be initialized with unfavourable fees for users either by mistake or by a malicious administrator.

Recommendation

To ensure the security and integrity of the contract, it is highly recommended to implement a comprehensive validation routine for all parameters during both the contract initialization and updating functions. This will help prevent any potential vulnerabilities or erroneous values from being accepted.

Furthermore, it is advisable to set strict hardcoded limits for the provided fee values. By establishing predefined boundaries, you can ensure that the fees remain within safe and reasonable ranges, minimizing the potential for unfavorable or maliciously set fees.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Bucket's Aave Liquidity May Be Temporarily Blocked

Medium | RES-PRMX-TRP06 | Transaction Ordering | Acknowledged

Code Section

- contracts/Bucket/Bucket.sol#L307

Description

The majority of withdrawal functions in `Bucket.sol` are safeguarded by the liquidity mining period. However, the `withdrawByAdminFromAave` function can be executed at any time. Withdrawing all collateral from the AAVE pool may cause a temporary lock on the bucket's Aave liquidity.

- aave-v3-core/contracts/protocol/libraries/logic/SupplyLogic.sol#L134

The impact of this issue has been lowered as the function can only be invoked by `SMALL_TIMELOCK_ADMIN`.

Recommendation

It is recommended to add an appropriate safeguarded mechanism to check the liquidity mining period when calling the `withdrawByAdminFromAave` function or keep some buffer so not everything can be withdrawn.

Status

The issue was acknowledged by Primex's team. The development team stated "The method "returnLiquidityFromAaveToBucket" allows for the return of lenders' liquidity from the Aave pool to the Bucket before the end of the accumulation period. This liquidity can then be used in the regular manner defined by the Bucket, serving as an additional security measure.".

Possible To Activate Bucket Without Adding It

Medium | RES-PRMX-TRP07 | Data Validation | Resolved

Code Section

- contracts/PrimexDNS/PrimexDNS.sol#L89-92

Description

The function `activateBucket()` allows the activation of a bucket without having been added with `addBucket()` in the first place. The validation on the `activateBucket()` function is based solely on the `currentStatus` of the bucket, which by default is `Inactive` for buckets that have not been added yet. This effectively means that a less privileged admin `SMALL_TIMELOCK_ADMIN` may effectively obtain the same result of adding and activating a bucket by simply calling the function `activateBucket()`.

Having a bucket activated with a corresponding name but not with a corresponding address can result in multiple behaviors across the various function calls throughout the protocol.

The intended workflow should be having the function `addBucket()` be executed first by the `BIG_TIMELOCK_ADMIN`, and then activate and freeze the bucket at will with the respective functions.

Recommendation

It is recommended to implement a validation to ensure that only buckets that have addresses set up can be activated.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Rewards Can Be Claimed When LiquidityMiningRewardDistributor Is Paused

Medium | RES-PRMX-TRP08 | Business Logic | Resolved

Code Section

- contracts/LiquidityMiningRewardDistributor/LiquidityMiningRewardDistributor.sol#L127

Description

The function `claimReward()` does not allow rewards to be claimed when the contract is paused, due to the usage of the `whenNotPaused` modifier. However, this is not the only function that is able to claim liquidity mining rewards. The function `reinvest()`, which is callable by buckets and can be triggered by the average user, can be used to claim liquidity mining rewards under certain conditions. When the `LiquidityMiningRewardDistributor` contract is paused, no rewards should be claimed one way or another, assuming that an emergency is in place.

Recommendation

It is recommended to move the modifier `whenNotPaused` from the function `claimReward()` to the internal function `_claimReward()`.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Possible To Use Deposit Owned By Other Users On depositPrologantion()

Low | **RES-PRMX-TRP09** | Data Validation | **Resolved**

Code Section

- contracts/PToken/PToken.sol#L108-117

Description

The function `depositPrologantion()` does not validate whether the `_depositId` belongs to the `msg.sender`. As such, a user can call this function erroneously and trigger a prolongation of his own deposit by an incorrect and/or unexpected amount.

Recommendation

It is recommended to validate whether the `_depositId` belongs to the msg.sender before attempting to prolong the deadline of the deposit.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Missing supportsInterface() Validation Of liquidityMiningRewardDistributor

Low | RES-PRMX-TRP10 | Data Validation | Resolved

Code Section

- contracts/Bucket/Bucket.sol#L17-85

Description

The `initialize()` function does not validate whether the argument `LMparams.liquidityMiningRewardDistributor` is a valid contract that supports the interface `ILiquidityMiningRewardDistributor`.

Recommendation

It is recommended to validate all address parameters to the respective intended interfaces, whenever possible.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Missing supportsInterface() Validation Of _bucketImplementation

Low | RES-PRMX-TRP11 | Data Validation | Resolved

Code Section

- contracts/Bucket/BucketsFactory.sol#L31-46

Description

The constructor does not validate whether the argument `_bucketImplementation` is a valid contract that supports the interface `IBucket`.

Recommendation

It is recommended to validate all address parameters to the respective intended interfaces, whenever possible.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Missing Validation Of Zero Amount On batchBurn()

Info	RES-PRMX-TRP12	Data Validation	Acknowledged

Code Section

- contracts/DebtToken/DebtToken.sol#L131-170

Description

The function `batchBurn()` does not validate whether the `_amounts` array contains elements with 0 amount, allowing transactions to take place and events to be emitted that do not change the state of the blockchain. This fact not only incurs unnecessary gas costs, but also fills the blockchain with unnecessary transactions,

Recommendation

It is recommended to validate the elements of `uint` arrays to verify whether unnecessary transactions will be made, e.g when amount is 0.

Status

The issue was acknowledged by Primex's team. The development team stated "The intention was to ensure that batch closing was made as cost-effective as possible in terms of gas and that unnecessary checks were minimized. It is assumed that the number of margin positions with zero debt will be considerably lower compared to the total number of positions processed by Keepers. This will result in reduced costs for them.".

Usage Of Unlimited Approve

Info	RES-PRMX-TRP13	Code Quality	Resolved

Code Section

- contracts/ActivityRewardDistributor/ActivityRewardDistributor.sol#L41
- contracts/Bucket/Bucket.sol#L70
- contracts/LiquidityMiningRewardDistributor/LiquidityMiningRewardDistributor.sol#L52
- contracts/SpotTradingRewardDistributor/SpotTradingRewardDistributor.sol#L49

Description

The inclusion of unlimited approval in contracts increase potential attack surface.

Recommendation

It is recommended to usage limited approval mechanism. This approach allows users or contracts to grant approval for specific actions or within predetermined limits. By restricting the scope of approval, the potential attack surface is significantly reduced.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Different Versions Of canBeFilled() Lead To Undefined Behaviour

Info | **RES-PRMX-TRP14** | Business Logic | **Resolved**

Code Section

- contracts/conditionalManagers/LimitPriceCOM.sol#L38
- contracts/conditionalManagers/LimitPriceCOM.sol#L138

Description

The function `canBeFilled()` implements several versions used throughout different contracts of the Primex protocol. Using different versions of code for the same intended behaviour may lead to unexpected and/or inconsistent results.

Recommendation

It is recommended to adapt the code logic to use only one version of function that implements the intended behaviour.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Unnecessary Validation Of _params On createBucket()

Info | RES-PRMX-TRP15 | Gas Optimization | Resolved

Code Section

- contracts/Bucket/BucketsFactory.sol#L52-56

Description

The function `createBucket()` correctly validates `_params.positionManager` and `_params.reserve`. However, this validation is unnecessary since the same validation is being made inside the contract `Bucket`, therefore leading to a duplication of code and the increase of gas costs.

Recommendation

It is recommended to remove duplicate code in order to save gas.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Conditions On supportsInterface() Can Be Switched To Save Gas

Info | **RES-PRMX-TRP16** | Gas Optimization | **Resolved**

Code Section

- Not defined

Description

Most `supportsInterface()` functions across the Primex protocol validate the base-most `interfaceId`. Due to both the inheritance nature of the Solidity language, and the short-circuiting capability of its logical operators, testing the base-most interface first is not optimal to save gas.

Recommendation

It is recommended to switch the conditions on the function `supportsInterface()` so that the child-most `interfaceId` is verified first.

Status

The issue has been fixed in bb95a3eeb398934ee4cda502dde150dfeeba482d.

Proof of Concepts

RES-01 Possible To Drain Treasury On closePositionByCondition()

KeeperRewardDistributor.test.js (added lines):

```
it("Exploit1 - Should claim reward on position with reason CLOSE_BY_TRADER", async
↪  function () {
  const conditionIndex = 0;

  const stopLossPrice = BigNumber.from(price).sub("1").toString();
  const takeProfitPrice = BigNumber.from(price).add("1").toString();

  OpenPositionParams.closeConditions = [
    getCondition(TAKE_PROFIT_STOP_LOSS_CM_TYPE,
    ↪  getTakeProfitStopLossParams(takeProfitPrice, stopLossPrice)),
  ];
  await positionManager.connect(trader).openPosition(OpenPositionParams, { value:
  ↪  feeAmountInEth });

  const spendingLimits = {
    maxTotalAmount: MaxUint256,
    maxAmountPerTransfer: MaxUint256,
    maxPercentPerTransfer: parseEther("1"),
    minTimeBetweenTransfers: 1,
    timeframeDuration: 60 60 24,
    maxAmountDuringTimeframe: MaxUint256,
  };
  await
  ↪  treasury.connect(deployer).setMaxSpendingLimit(KeeperRewardDistributor.address,
  ↪  PMXToken.address, spendingLimits);
  await
  ↪  treasury.connect(deployer).setMaxSpendingLimit(KeeperRewardDistributor.address,
  ↪  NATIVE_CURRENCY, spendingLimits);
  await PMXToken.transfer(treasury.address, parseEther("100000000"));
  await deployer.sendTransaction({
    to: treasury.address,
    value: parseEther("1000"),
  });

  await positionManager
    .connect(trader)
    .closePositionByCondition(0, trader.address, routesForClose, conditionIndex,
    ↪  "0x", CloseReason.CLOSE_BY_TRADER);

  const ethBalanceBefore = await provider.getBalance(trader.address);
  const pmxBalanceBefore = await PMXToken.balanceOf(trader.address);

  const { pmxBalance: pmxReward, nativeBalance: nativeReward } = await
  ↪  KeeperRewardDistributor.keeperBalance(trader.address);
  const tx = await KeeperRewardDistributor.connect(trader).claim(pmxReward,
  ↪  nativeReward);
```

```
  await expect(tx).to.emit(KeeperRewardDistributor,
  ↪  "ClaimFees").withArgs(trader.address, NATIVE_CURRENCY, nativeReward);
  await expect(tx).to.emit(KeeperRewardDistributor,
  ↪  "ClaimFees").withArgs(trader.address, PMXToken.address, pmxReward);

  const receipt = await tx.wait();
  const transactionCost = receipt.gasUsed * receipt.effectiveGasPrice;
  const ethBalanceAfter = await provider.getBalance(trader.address);
  const pmxBalanceAfter = await PMXToken.balanceOf(trader.address);

  ↪  expect(ethBalanceAfter.sub(nativeReward)).to.be.equal(ethBalanceBefore.sub(transactionCo
  expect(pmxBalanceAfter.sub(pmxReward)).to.be.equal(pmxBalanceBefore);

  const { pmxBalance: pmxRewardAfter, nativeBalance: nativeRewardAfter } = await
  ↪  KeeperRewardDistributor.keeperBalance(trader.address);
  expect(pmxRewardAfter).to.be.equal(0);
  expect(nativeRewardAfter).to.be.equal(0);

  const { pmxBalance: totalPmxBalance, nativeBalance: totalNativeBalance } = await
  ↪  KeeperRewardDistributor.totalBalance();
  expect(totalPmxBalance).to.be.equal(0);
  expect(totalNativeBalance).to.be.equal(0);
});
```

RES-08 Rewards Can Be Claimed When LiquidityMiningRewardDistributor Is Paused

Bucket.test.js (added lines):

```
it("Exploit2 - depositFromBucket works when LiquidityMining is paused", async
↪  function () {
  await testTokenA.connect(lender).approve(bucketSame.address,
  ↪  liquidityMiningAmount);
  await bucketSame.connect(lender).deposit(lender.address, liquidityMiningAmount,
  ↪  0);
  expect((await
  ↪  bucketSame.getLiquidityMiningParameters()).isLaunched).to.equal(true);

  await LiquidityMiningRewardDistributor.pause();

  await expect(bucket.connect(lender).depositFromBucket(bucketSameName,
  ↪  swapManager.address, [], 0)).to.emit(
    LiquidityMiningRewardDistributor,
    "ClaimedReward",
  );
});
```